Software Synthesis via Domain-Specific Software Architectures

William M. Waite and Anthony M. Sloane

CU-CS-611-92 September 1992

University of Colorado at Boulder

Technical Report CU-CS-611-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **SEP 1992**	2. REPORT TYPE	3. DATES COVERED **00-00-1992 to 00-00-1992**
4. TITLE AND SUBTITLE **Software Synthesis via Domain-Specific Software Architectures**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Department of Computer Science,University of Colorado,Boulder,CO,80309**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**18**	

Software Synthesis via Domain-Specific Software Architectures

William M. Waite and Anthony M. Sloane

CU-CS-611-92 September 1992

Software Synthesis via Domain-Specific Software Architectures

William M. Waite and Anthony M. Sloane

September 1992

Abstract

Current software engineering practice concentrates on improving the process by which a programmer develops a solution from the description of a problem; we describe a new paradigm for software synthesis based on Domain-Specific Software Architectures (DSSAs) that eliminates this process entirely. A DSSA provides an overall software design that solves a whole class of problems in a broad area. It focuses the designer's attention on the unique requirements of the current problem, suppressing those that are common to all problems of the type addressed by that DSSA. To use the DSSA approach, a software engineer provides a description of the unique requirements of a particular problem. A solution to that problem is then generated according to the DSSAs overall design by a system that implements the DSSA. Problem descriptions are checked for consistency by the system, and the generated software is guaranteed to solve the problem described.

We briefly describe how we have used the DSSA approach to build Eli, a system for compiler construction. Generalizing from Eli, we identify requirements that the implementation of any DSSA should satisfy: 1) incorporation of a manufacturing language to describe the incremental derivation of software objects with architecture-based error reporting, 2) incorporation of an authoring language to allow on-line access to documentation and system components, and 3) the ability to incorporate externally developed tools and export constructed programs.

1 Introduction

A promising method for dealing with the difficulty of engineering complex pieces of software is to synthesize programs from specifications. A programmer writes a description of the problem that they want to solve and the synthesis system translates that description into an implementation.

Specification languages range from being very close to implementation level to being concerned with very abstract concepts. The kind of specification language directly influences the dividing line between the work that must be done by the programmer and that which must be done by the system. A very abstract specification language pushes work into the system and makes it correspondingly harder to build. Less abstract notations can be implemented more easily but require the programmer to deal with low-level details that may be irrelevant to the particular problem being solved.

One way to build more abstraction into software synthesis systems is to work within a particular application domain and to identify commonality between applications in that domain. Common functions can have specification languages designed specifically for them and can be implemented in the same or similar ways by all programs that require them. Functionality that is not common can be handled in a more ad-hoc manner.

The relationships between functions in programs for a software domain define a *Domain-Specific Software Architecture (DSSA)* [10]. In order to design a program within the domain a user maps their problem into functionality embodied in the architecture.

A DSSA is an abstract structure. To achieve software synthesis for a given DSSA it is necessary to implement that architecture in a system—a *Domain-Specific Software Constructor (DSSC).* A DSSC provides manufacturing processes that take specifications of components of the architecture and implement them using appropriate methods. The user views a DSSC through the embodied architecture. Arbitrarily complex construction techniques can be totally hidden.

This paper begins by motivating and describing the DSSA approach to software development by considering in detail the kinds of problems that it is designed to solve. A brief discussion of our experiences in creating a DSSC for compilers leads into a presentation of requirements that an implementation of a DSSA must satisfy and how we satisfied them. We conclude with some pointers to future work that is needed to advance DSSA-based software construction.

2 Domain-Specific Software Architectures

Most problems are too complex to be solved directly. Instead, they are decomposed into simpler problems. These simpler problems may be decomposed in turn until some set of elementary components is reached. For example, creating a compiler for a new programming language/target machine pair can be recognized as an instance of the compiler construction problem. We know that compiler construction problems can be decomposed in a certain way into simpler problems like scanning, parsing, name analysis, code generation, and so forth. A problem that is normally approached in this way will be called a *composite problem.*

2.1 Understanding Composite Problems

Consider the following composite problem: "obtain an artificial environment for some human activity". Obtaining an artificial environment for the day-to-day life of a family is one instance of this problem, obtaining an artificial environment for carrying out the fabrication of automobiles is another. The former instance is solved by a building called a *house*, while the latter instance is solved by a building (or collection of buildings) called a *factory.* In each case, however, it is possible to generate a solution in the form of one or more buildings.

We decompose this composite problem into subproblems like the type and amount of space needed for the activity, the appropriate climate within the environment, and the exterior appearance of the building. A particular instance of the composite problem (e.g. an environment for the day-to-day activities of a family of four) yields particular instances of the component problems (e.g. three bedrooms, two baths, a kitchen/dining area and a living room). To solve the composite problem, we specify all of the subproblem instances to a building contractor. After some expenditure of time and money the solution (a building suitable for this activity) appears.

In this example, the *user* is the person who needs to obtain the environment for the day-to-day activities of a family of four. They describe that instance of the composite problem by providing *specifications* of component problem instances to the *constructor* (a building contractor). The constructor, in turn, generates the solution (builds the house). Finally, the user uses the generated solution to solve the original problem.

It is not necessary to hire a building contractor to solve an instance of the problem we have been discussing. Many people design and build their own houses, possibly using subcontractors for carrying out specific tasks. The tradeoffs are obvious: A person building their own house must have a much deeper understanding of building construction than one who uses a building contractor. They must spend much more of their own time in the process (although the total elapsed time may not differ very much), but they have more control over the details of the finished product. The quality of the finished product may be higher or lower, depending upon the knowledge and skill of the person.

A building contractor embodies a collected understanding of the composite problem. By replacing incompetent workers by skilled workers, the building contractor maintains a high level of competence in fabricating the parts of a building. It is this knowledge and competence that one accesses when one uses the contractor. When we create a constructor to generate solutions to a composite problem, it must embody our collected knowledge of that composite problem and its solution.

An understanding of a composite problem and its solution involves three kinds of knowledge:

- How to decompose the composite problem into component problems

- How to solve each of the component problems
- How to compose the individual solutions of the component problems into a solution of the composite problem

2.2 Solving Composite Problems with DSSAs

A Domain-Specific Software Architecture embodies a large fraction of the knowledge required to solve a composite problem. It gives an overall design that will satisfy a whole class of requirements. It embodies a particular decomposition of problem instances in its domain, support for the solution of component problems, along with the process for combining solutions for the components into a solution for the total problem instance. This approach is only useful in cases where a broad problem area has been recognized, general requirements formulated, and a feasible problem-solving strategy developed.

2.2.1 Problem Decomposition

A composite problem is decomposed by a DSSA into a set of elementary component problems that are solved by *atomic actions.* Typical atomic actions in the construction of a building might be "install brick veneer" and "install electric wiring". Examples from compiler construction are "generate an operator identification module" and "generate an attribute evaluation module". We will explore the characteristics of atomic actions in the next subsection; for the purposes of this subsection it is sufficient that the architecture has access to some specific set of atomic actions.

The examples of atomic actions given in the previous paragraph illustrate two kinds of variability in problem decomposition: variability due to problem instance and variability due to user understanding.

Different instances of the composite problem may decompose into different sets of component problems, and therefore their solutions may involve different sets of atomic actions. For example, a cinder block building does not require a brick veneer, and therefore its construction does not involve the atomic action "install brick veneer". Similarly, if none of the operators of a source language is overloaded then a compiler for that language has no need of an operator identification module.

When the decomposition of the composite problem instance includes a particular subproblem, a user may or may not be interested in describing the solution of that subproblem. The user's interest will depend on how knowledgeable they are and how critical to their satisfaction with the final product they perceive that subproblem to be. A person specifying a building might say nothing about the electric wiring, thus obtaining a default solution determined by local building codes. On the other hand, a person with woodworking tools might specify additional outlets and higher-capacity circuits in some areas. Similarly, the standard time/space tradeoff for an attribute evaluator [7] will suffice for most compiler designers, but a modification might be desirable under certain circumstances.

Specification of a set of elementary actions often involves unproductive redundancy. Two of the elementary actions required to construct a compiler are *generate a scanner* and *generate a parser*. Each of these actions depends upon a list of the basic symbols of the language. An architecture for the compiler construction problem can generate these two lists from information contained in a non-redundant specification of the particular instances of the scanning and parsing problems to be solved.

In general, the problem decomposition made visible to the user by an architecture differs from the decomposition that a system implementing that architecture actually employs to solve the problem. The *visible decomposition* should provide orthogonal specifications for subproblem instances, and match those specifications as closely as possible to the user's experience. This is possible because the architecture considers the composite problem as a single entity rather than merely the sum of its components. Specifications of component problem instances are different views of the composite problem instance, each concentrating on a different aspect. The system integrates these views to obtain its solution, automatically combining the different perspectives into a single result.

2.2.2 Solution of Component Problems

A DSSA focuses the user's attention on the requirements and design decisions that are unique to the current problem, having already provided solutions to those that are common to all problems of the type addressed by that DSSA. A DSSC implementing the DSSA accepts descriptions of those requirements and design decisions, and combines them with its understanding of the domain's common problems to produce software that will carry out the described task. Requirements and design decisions can be thought of as specifications of instances of subproblems of the problem to be solved. There are three basic ways in which a user might specify an instance of a subproblem:

- *By analogy*: "This problem is the same as problem X."
- *By description*: "This is a problem of class Y, and is characterized as follows..."
- *By solution*: "Here is a program to solve this problem."

To support specification by analogy, the DSSC must provide a library of problem solutions. A user must have sufficient understanding of the problem to recognize that it has been solved before and to find the solution in the library. To support specification by description, the constructor must provide a collection of tools that recognize notations specific to appropriate problems. A user must not only be able to recognize the kind of problem, but also understand the notation used to characterize problems of that kind. To support specification by solution, the constructor must be able to accept arbitrary program fragments and incorporate them into the software being constructed. A user must not only be able to recognize the kind of problem, but also solve that problem completely.

Specifications by analogy and by description are both forms of re-use: A specification by analogy re-uses a particular solution, while a specification by description re-uses a problem-solving method. In each case, re-use simplifies the specification by allowing much to be omitted. A DSSA provides leverage in direct proportion to the set of problems for which it embodies solutions and problem-solving methods. There are very few domains, however, for which solutions and problem solving methods are known for all instances of all component problems. Almost invariably, therefore, some problem instances must be specified by solution (at least during the early phases of DSSA development).

3 A DSSC for Compiler Construction

An excellent example of a DSSA is the architecture for compilers that has been developed over the past twenty-five years. That architecture, which is described with minor variations in any textbook on compiler construction, satisfies the requirements for virtually every language/machine translation in use today. We have implemented it as a DSSC called Eli that runs on many UNIX[1] systems [6].

Eli has been used successfully by a number of groups to develop compilers for a variety of languages. Among these projects are a commercial source-to-source translator for FORTRAN, a research compiler for a version of C with constructs to support parallel numeric algorithms, and a translator for the Icon programming language. Efforts currently in progress include a translator for PSDL (a specification language for real-time systems) at the Naval Postgraduate School, a processor for VDM specifications at the Technical University of Delft, and an analyzer for finite-element data at Carleton University. The performance of Eli-generated code is quite close to that of its hand-coded equivalent (the generated Icon translator, for example is about 15% slower than the equivalent hand-coded program distributed by the developers of Icon).

Variability due to problem instance is handled by a sophisticated planning mechanism by which Eli determines the appropriate decomposition from the set of specifications provided. Standard solutions for the component problems of source code input and identifier encoding are typical of the support Eli provides for variability due to user understanding. Most users are not interested in describing their instances of these subproblems, but Eli allows easy overriding of the standard solutions to meet special requirements. The developer of the FORTRAN source-to-source translator took advantage of this variability to handle the peculiar line structure of FORTRAN.

Redundancy in specifications is avoided by deriving inputs to standard tools like the parser generator and scanner generator from diverse sources. The user is presented with a coherent model of the compilation problem, stated in terms of orthogonal subproblems. An extensive library allows many of the subproblem instances to be specified by analogy. Others are specified by description, using standard mechanisms like context-free grammars and attribute grammars to characterize the member of the subproblem class. Special languages have been provided to characterize instances of some classes of subproblems, such as that of overload resolution [8].

[1]UNIX is a registered trademark of UNIX System Laboratories, Inc.

Subproblem specification by solution is also supported by Eli. This method is used primarily for tasks like sophisticated optimization, where the current state of the art precludes specification by analogy or description. It is also important, however, when some subproblem has no known analogs and does not belong to a recognized class.

4 Realizing DSSAs

One motivation for our work on Eli has been to explore implementation techniques that allow us to realize DSSAs in DSSCs. Compiler construction has proven to be an excellent setting for experimentation. This section presents a set of requirements on an implementation mechanism that we have identified through our work with Eli.

When we consider DSSA implementation mechanisms, we need to distinguish a new role—that of the *implementor*. An implementor is the person who creates a DSSC, whereas the user is the person who uses that system to create solutions to composite problems.

Recall that a DSSA embodies understanding of how a program solving a particular instance of a composite problem is manufactured from a specification of that problem instance. It also guides the user in decomposing the problem into component problems, and may further decompose those components. Ultimately, a set of elementary component problems is reached. Each elementary component problem is assumed to have a solution that can be obtained by invoking a tool, extracting a library component, or incorporating a user-supplied component.

In practice, the manufacturing process embodied in a DSSC is complex. It involves a large number of steps, and carrying it out is time-consuming. User decomposition of the composite problem is also error-prone. Tool invocation may therefore lead to error reports, but because the tools are concerned only with elementary component problems these reports are not couched in terms of the composite problem. They must be restated by the system so that the user can repair their decomposition.

These properties of DSSCs can be used to develop requirements for suitable implementation mechanisms. Each subsection of this section discusses one of the properties mentioned above and explains the requirements arising from it.

It is important to note that a DSSA should *not* be implemented via a general-purpose programming environment like Field [9] or Forest [5]. These systems support general collections of tools interacting in user-defined ways, whereas a DSSC supports a specific set of tools interacting in a way defined by the implementor. It appears to the user as a single entity with embedded knowledge about the process being carried out. A DSSC might very well be one of the tools appearing in the general-purpose programming environment, but it would not *be* the environment.

4.1 Embody a Manufacturing Process

We know that a DSSC must accept specifications defining a composite problem instance, determine what set of atomic actions must be performed, create specifications to guide those atomic actions, and compose the results into a solution for the defined problem instance. This entire collection of actions constitutes the manufacturing process. Although each constructor carries out a different manufacturing process, the steps required fall into well-defined patterns.

In order to create a constructor, the implementor must describe the manufacturing process they intend to carry out. The atomic operations of this manufacturing process involve creation of objects on the basis of information contained in other objects.

> **Requirement:** A *manufacturing language* describing how to derive software objects from other software objects must be available.

We have seen that, depending on the particular instance of a problem being described, it may be necessary to make small changes in the manufacturing process. The implementor's description of the process must define the conditions under which such variations occur. Some of the variations will require specific instructions from the user, but others should be deduced from the definitions of subproblems.

> **Requirement:** The manufacturing language must be capable of describing different derivations, based on the form and content of the objects being processed.

4.2 Guide the User in Problem Decomposition

Users of a DSSC will come from a variety of backgrounds with varying levels of understanding of the composite problem the system solves. Therefore they will need guidance to know which component problems should be specified and which should not. Because the underlying DSSA provides a decomposition of the composite problem that is not simply the collection of elementary subproblems, the user guidance cannot be obtained automatically from the manufacturing process.

The implementor must produce text describing decomposition strategies. Most problem classes are rather broad, and therefore the characterizations of problem instances will vary. The guidance one would give to someone specifying a stadium would be rather different from the guidance given to the specifier of a mountain shack. Thus the text should be structured so that a user interested in a particular kind of guidance need not scan irrelevant material.

> **Requirement:** An *authoring language* describing structured on-line documentation must be available.

Users with special requirements will need to modify default specifications. For example, a user constructing a compiler for FORTRAN must be able to replace the default processing of line boundaries with a special one that recognizes the peculiar line structure of FORTRAN. Such modifications are often merely small perturbations of the default specification, but it is impossible to predict all of them a priori.

The implementor must include complete descriptions of default specifications in the decomposition strategy documentation. Since the desired modifications are usually small, the implementor should also provide access to machine-readable text of these default specifications.

> **Requirement:** The authoring language must be capable of describing access to system components, permitting the reader to include a modified copy in their own specifications.

4.3 Accept Off-the-Shelf Tools

One component of a compilation problem is *scanning*—the process of recognizing sequences of characters as instances of source language basic symbols. Scanning is also a component of the bibliographic search problem [1], the problem of correcting spelling errors in text [2], and many similar problems.

Tools implementing solutions to such common problems are often available "off-the-shelf". An implementor should employ such existing processors whenever possible in order to shorten the development time and improve the reliability of the final product. This means that the construction techniques must accommodate the disparate interface requirements of components that were not originally designed to work together. (This flexibility also makes it easy to replace a tool when a better one becomes available.)

> **Requirement:** Each atomic action may be implemented by an arbitrary collection of programs for which no source code is available.

4.4 Provide Incremental Manufacture

Use of a DSSC to obtain the solution of a composite problem involves the familiar modify-generate-test cycle found in any programming task. The process of manufacturing the solution is, however, far more complex than the process of compiling a single program. Specifications are also less monolithic than single routines, so a random change affects a much smaller part of the process. This means that incremental manufacture is a very useful technique for reducing the overall processing time.

> **Requirement:** The manufacturing process must retain a cache of intermediate objects and re-derive objects only when necessary.

4.5 Refer Error Reports to User Views

Every tool reports errors in terms of the conceptual framework in which it was built. Thus when we invoke a tool, that tool will report any errors in terms of a conceptual framework that lies outside of the one defined by the processor. Perhaps such error reports will be understandable to the user because their conceptual framework is a familiar one. In general, however, they will *not* be understandable.

The implementor of a DSSC must have enough understanding of the tools to know what their error messages mean in terms of their inputs. This understanding, combined with an understanding of the inputs in terms of the overall architecture, allows the implementor to express the error in terms of the conceptual framework of the constructor itself, which must be known to the user. An appropriate message text can thus be manufactured from the information provided by the tool.

Requirement: Arbitrary manufacturing steps may be applied to error reports.

4.6 Export constructed program

Many programs are used in situations far from their development environment. Programs constructed using a DSSC are no exception. It is unreasonable to expect every user of a program to install the DSSC that constructed the program in order to be able to execute it. Once implementations for each of the program's components have been constructed there should be no dependence on the constructor. By extracting the component implementations all ties to the constructor are broken and the program can be treated like any other. In this way a DSSC can be used in a similar fashion to more familiar software constructors such as compilers.

Requirement: It must be possible to export constructed programs in order to build and execute them independently of the constructor.

4.7 How Eli Satisfies the Requirements

As a concrete example of how these requirements can be satisfied, we briefly mention the techniques used in Eli.

The manufacturing language used by Eli is that of Odin, an expert system whose domain of expertise is the satisfaction of complex user requests [4]. The relationships among tools defined by the software architecture is expressed in a *derivation graph*, a data structure ideally suited to describing such relationships. Each node of the derivation graph is implemented by an arbitrary script, and the combination of inferencing on the graph and execution of the scripts allows the implementor of the DSSC to easily encode very complex processes. The use of arbitrary scripts allows off-the-shelf tools to be seamlessly integrated and Odin's maintenance of the status of objects automatically supports the rederivation of objects only when necessary.

By combining the expert system with Texinfo—an existing hypertext facility[3]—Eli makes it possible to "backtrack" error reports to the user-level documentation and provide context-dependent on-line help. The hypertext can also provide discussions of design strategy with sample specification fragments that can be automatically incorporated in the user's problem description.

Eli exports constructed programs as C source files which can be transported to another system, configured for that system, built and executed. Once the sources have been generated Eli is no longer required.

The use of Odin has allowed Eli to evolve gracefully. Tools have been replaced with more powerful or more efficient versions. In many cases it was possible to update the system with little or no change to the specifications supplied by users. New capabilities can be added easily, sometimes in as little as a couple of hours.

5 Conclusions and Future Work

Domain-Specific Software Architectures arise naturally out of a consideration for the structure of solutions to complex problems: decomposition of the problem, specification of subproblem instances, and composition of subproblem solutions. Re-use is supported by allowing the user to specify problems in terms of identical ones (analogy) or similar ones (description). Less well understood subproblems can be described completely (solution). By allowing a user to concentrate on the unique aspects of their particular problem, DSSAs effectively reduce the required level of sophistication. We have observed significant advantages for new users.

Our experience with Eli has illuminated a number of important requirements that should be addressed by any attempt to implement a DSSA. Sophisticated build processes must be supported in a flexible and incremental manner. On-line documentation should be incorporated to present the domain architecture to users with differing levels of sophistication. Of particular importance is the ability for users to be able to access default specifications, modify them and include them in their own specifications.

A DSSC cannot be economically created in a vacuum. An implementation mechanism must allow outside expertise to be incorporated via off-the-shelf tools. Because such tools may have been created without any knowledge of the use to which they will be put in the DSSC it must be possible for the implementor to process any tool reports to match them to the user's overall view of the system. There is also a necessary interface to the outside world after a program has been constructed by the DSSC. Most users of a DSSC are not willing to require that all end-users of their programs install the DSSC in order to run those programs. Thus it must be possible to export programs after construction.

It is important to note that the development of Eli was incremental, and that it continues today. The techniques we have described in this paper support the notion of an evolving DSSC, which initially handles a small part of the construction problem. When the burden of one part of the process is taken over by the DSSC, users begin to note the repetitive nature of other parts of the problem and develop ways to automate those. Those problems are then taken over by the DSSC and still others become noticeable. The fact that the structure of the manufacturing process is described at a high level and the tools are independent of one another makes it very simple to accommodate such growth.

Some of our current work concerns a problem area that has received little attention in the context of DSSAs: *execution monitoring* (debugging, profiling and performance measurement). It is impossible to build a DSSC for a non-trivial domain that guarantees the correctness and adequate performance of all constructed programs. We are currently extending the notion of a DSSA to include execution monitoring. Of particular importance is ensuring that these extensions do not affect the ability of implementations to satisfy the requirements of the previous section; it is essential that off-the-shelf tools be supported, for example.

We are also exploring architectures for the problem of constructing a DSSC. A designer using a DSSC that implemented such an architecture would need a thorough understanding of their problem domain, but only minimal knowledge about DSSA implementation. This work is in its early stages; the present paper summarizes our current ideas, and we have an experimental implementation of the task of deriving support for error report transformations.

The Eli system has demonstrated that the requirements stated in this paper can be satisfied in a sophisticated DSSC built using current technology. The combination of Odin and Texinfo running on a UNIX system provides a framework, and various home-grown and off-the-shelf tools and library modules flesh out that framework for a specific domain. Leverage is obtained relative to a more traditional approach by concentrating the user's attention on the aspects of their problem that distinguish it from other problems in the same class, increasing the number of subproblems that can be solved by analogy or by description, and automating the process of composing subproblem solutions.

Acknowledgements

This work was supported by the US Army Research Office under grants DAAL03-92-G-0158, DAAL03-91-G-0203 and DAAL03-86-K-0100.

References

1. AHO, A. V. AND CORASICK, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM 18*, 6 (June 1975), 333–340.
2. BENTLEY, J. A Spelling Checker. *Communications of the ACM 28*, 5 (May 1985), 456–462.
3. CHASSELL, R. J. AND STALLMAN, R. M. *Texinfo: the GNU Documentation Format.* Free Software Foundation, Feb. 1992.
4. CLEMM, G. AND OSTERWEIL, L. A Mechanism for Environment Integration. *ACM Trans. on Programming Languages and Systems 12*, 1 (Jan. 1990), 1–25.
5. GARLAN, D. AND ILIAS, E. Low-Cost, Adaptable Tool Integration Policies for Integrated Environments. *ACM SIGSOFT Software Engineering Notes 15*, 6 (Dec. 1990), 1–10.
6. GRAY, R. W., HEURING, V. P., LEVI, S. P., SLOANE, A. M. AND WAITE, W. M. Eli: a complete, flexible compiler construction system. *Communications of the ACM 35*, 2 (Feb. 1992), 121–131.
7. KASTENS, U. Lifetime Analysis for Attributes. *ACTA Informatica 24* (1987), 633–651.
8. PERSCH, G., WINTERSTEIN, G., DAUSMANN, M. AND DROSSOPOULOU, S. Overloading in Preliminary Ada. *ACM SIGPLAN Notices 15*, 11 (Nov. 1980), 47–56.
9. REISS, S. P. Connecting tools using message passing in the Field environment. *IEEE Software 7*, 4 (July 1990), 57–66.
10. SOFTWARE ENGINEERING INSTITUTE, CARNEGIE MELLON UNIVERSITY. *Workshop on Domain-Specific Software Architectures.* Hidden Valley, PA, July 1990.